

Customer No. 20350
TOWNSEND and TOWNSEND and CREW LLP
Two Embarcadero Center, 8th Floor
San Francisco, California 94111-3834
(415) 576-0200

Attorney Docket No. 20133-000300US

"Express Mail" Label No. EL394878433US

Date of Deposit: July 24, 2000

ASSISTANT COMMISSIONER FOR PATENTS
BOX PATENT APPLICATION
Washington, D.C. 20231

Sir:

Transmitted herewith for filing under 37 CFR 1.53(b) is the

- ☒ patent application of
☐ continuation patent application of
☐ divisional patent application of
☐ continuation-in-part patent application of

EL394878433US

JC542 U.S. PTO
09/625226
07/24/00

Inventor(s)/Applicant Identifier: Paul Wensley and Richard T. Minner

For: METHOD AND SYSTEM USING NON-UNIFORM IMAGE BLOCKS FOR RAPID INTERACTIVE VIEWING OF DIGITAL IMAGES OVER A NETWORK

Enclosed are:

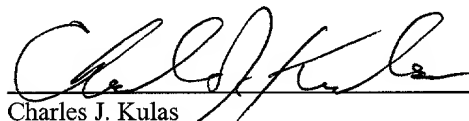
- ☒ 15 page(s) of specification
☒ 6 page(s) of claims
☒ 1 page of Abstract
☒ 7 sheet(s) of ☐ formal ☒ informal drawing(s).
☐ An assignment of the invention to _____
☐ A ☐ signed ☐ unsigned Declaration & Power of Attorney
☒ A ☐ signed ☒ unsigned Declaration.
☐ A Power of Attorney.
☐ A verified statement to establish small entity status under 37 CFR 1.9 and 37 CFR 1.27 ☐ is enclosed ☐ was filed in the prior application and small entity status is still proper and desired.
☐ A certified copy of a _____ application.
☐ Information Disclosure Statement under 37 CFR 1.97.
☐ A petition to extend time to respond in the parent application.
☐ Notification of change of ☐ power of attorney ☐ correspondence address filed in prior application.
☒ Microfiche Appendix

**In view of the Unsigned Declaration as filed with this application and pursuant to 37 CFR §1.53(f),
Applicant requests deferral of the filing fee until submission of the Missing Parts of Application.**

DO NOT CHARGE THE FILING FEE AT THIS TIME.

Telephone:
(415) 576-0200

Facsimile:
(415) 576-0300


Charles J. Kulas
Reg. No.: 35,809
Attorneys for Applicant

PATENT APPLICATION

**METHOD AND SYSTEM USING NON-UNIFORM IMAGE BLOCKS
FOR RAPID INTERACTIVE VIEWING OF DIGITAL IMAGES OVER A
NETWORK**

Inventor(s):

Paul Wensley
90 George Lane
Sausalito, CA 94965
A citizen of the United States of America

Richard T. Minner
2635 Napoli Court
Carmichael, CA 95608
A citizen of the United States of America

Assignee: Xippix, Inc.
80 Sir Francis Drake Blvd.
Larkspur, CA 94939

Entity: Small

004220 92252650

5

10

15

20

25

30

1

interactive computer use session: (1) the client computer itself; (2) the network connecting the client computer to the server computer; (3) the server computer itself.

Interactive browsing of digital media can be made quicker and more responsive to the user by increasing the speed with which any of these three components operates, or by decreasing the quantity of data that the components need to process.

In the prior art for interactive server-client image-viewing systems, two significant innovations have been the storage of pyramided images on the server, and the partitioning of such images into uniform-sized tiles. Pyramided images facilitate zooming. Uniformly-tiled images facilitate quick access and delivery of image segments by the server computer.

A pyramided image is one in which the original digital image is supplemented by a succession of lower-resolution subsampled versions of the same image. Typically, each successive subsampled image has one-half the linear (row and column) resolution of its predecessor, and hence one-quarter the areal resolution of its predecessor. Typically, the chain of subsampled images continues down to a final one which is sufficiently small to be suitable for use as a thumbnail image on a computer screen. Pyramided images are particularly valuable in a server-client image-viewing system, because they mean that the client computer does not have to do any of the computational work required for image zooming; image zooming becomes, for the client computer, a matter not of computing a downsampled image but instead of requesting the appropriate downsampled image portion from the server computer and displaying it when received.

A uniformly-tiled image is one in which the image is decomposed into a uniform grid of uniformly sized, normally square, tiles. Common tile sizes are 64 rows by 64 columns, or other squares whose linear dimension is a power of two. Once an image has been decomposed into tiles, the tiles are saved in a file format such that a tile is identifiable by its offset in rows and columns (of tiles) from the image origin, which, by convention in digital imaging systems, is at the top left of the image. Storing an image in uniform-tile format greatly simplifies, facilitates, and speeds up the task of fetching an image portion, normally a rectangular window, from inside of an image. For example, in the action of panning, the user of the computer-viewing system causes a rectangular viewing window to traverse a larger, mostly unseen, image. Each time the viewing window pans to a new location, the computer system must find the requisite rectangular portion of the image from within the file and deliver it to the portion of the computer

system that causes the image to be displayed. If the image is untiled, the task of finding a particular rectangular subportion of the image can be time-consuming; it can require stepping through all the pixels (picture elements) in the image up to the origin, and then to the opposite corner, of the rectangular subwindow. If, however images are uniformly
5 tiled, the computer system software can readily identify the set of tiles minimally necessary to envelop any given subrectangle of the image, and, since the tiles are indexed by their row and column numbers, the system software can readily access them and deliver them to this display system.

An image can be both pyramided and uniformly tiled. In this case, each
10 single resolution "layer" of the image, including the original full-resolution layer, is partitioned into image tiles of the same dimensions, for example, 64 rows by 64 columns. As the image pyramid progresses from one layer to a subsequent subsampled lower-resolution version of it, the overall size of a layer in pixels decrease by a factor of (approximately) four. ("Approximately" because of possible quantization roundoff.) But
15 the sizes of the tiles into which the layer is partitioned stay constant. Hence the number of tiles per layer decreases by a factor of approximately four.

Uniformly-tiled images permit a computer seeking a rectangular subwindow of an image file to have random access to the minimal set of tiles required to envelop that subwindow. In order for the computer to gain random access to the tiles, it
20 is critically important that the tiles fill a uniform regular grid over the image area rather than randomly or chaotically filling the image area. For example, the irregular area-filling tiling schemes of Roger Penrose (as described in U.S. patent 4,133,152) would not permit random access to rectangular image subportions.

Given a server computer with access to a set of pyramided, uniformly-tiled
25 images, a uniform-tile client-server image-browsing system as practiced in the prior art, is one in which the same tiles as are resident on the server computer are transmitted over the network to client computers, where they are displayed and are perhaps also cached for later reuse.

If such a system has a caching feature, it may operate as follows. When
30 the user of the client computer first displays an image, the client system requests from the server computer just those uniformly-shaped tiles necessary to render the newly requested portions of the image. The tiles so requested are placed in a cache, and the portions of them requisite for the current image display (cropped to the viewing window) are then displayed. The client computer's image tile cache is now seeded. Subsequently, when the

user pans or zooms the image, thus causing a new portion of the image to be needed for display, the client computer (a) determines the minimal set of tiles needed to render the new image portion; (b) determines which, if any, of the requisite tiles are contained in the client computer's tile cache; (c) determines, as the residual (a)-(b) the set of tiles, if any,
5 needed to be requested from the server; (d) requests the requisite tiles from the server; (e) on receipt of the requisite tiles from the server, adds them to the client-computer tile cache; (f) fetches from the client-computer tile cache the entire set of tiles required to render the new image portion, and causes them to be displayed, clipped to the current view window.

10 As image-viewing operations continue like this, data continues to be added into the client computer's tile cache. Consequently, the tile cache consumes progressively more of the client computer's memory. In order to limit the amount of memory used by the cache, procedures for systematic tile cache purging are incorporated into the client system. An upper bound for memory occupied by the cache is specified. When addition
15 of new tiles to the client cache would cause this upper bound to be exceeded, tiles are purged from the cache. A least-recently-used tile-purging scheme is effective for determining which tiles to purge.

In the uniform-tile client-server image-viewing system described above, several of the specific operations described are facilitated by the fact that the cache
20 maintained by the client is a regular grid of uniformly-sized tiles. In particular, the operations (a), (b), (e) and (f) described above are all made faster by the fact that the grid of tiles is uniform, so that a tile location can be immediately calculated from an image location, and tiles can be randomly accessed.

In interactive image browsing, the bottleneck to rapid image viewing in the
25 three-part client-network-server system is almost always the network. In particular, in almost all client-network-server systems for image browsing, an analysis of the system capabilities for a typical individual client-server session will find that the speed of image browsing by an individual user on his or her client computer could not be increased by increasing the capabilities of the client or server computer unless the network were made
30 to operate more efficiently.

In the uniform tile client server image viewing system described above, uniformly-sized tiles are used in all three components of the server-network-client system. Images are saved on the server in a uniform grid of tiles; each segment of image data transmitted over the network constitutes an identically-sized tile from the uniform

grid, or a set of such tiles; the client computer's cache of image portions is a cache of a set of identically-sized tiles from the uniform grid, each of the same specification as the identically-sized tiles stored on the server computer and transmitted over the network.

However, the choice of the use of a grid of uniformly-sized tiles, though
5 beneficial for image access efficiency on the server computer and client computers, is suboptimal for network efficiency.

The choice of a grid of uniformly-sized tiles forces the system to transmit a significant portion of unneeded pixels along with the needed pixels over the network. For example, suppose that all tiles are of size 64 rows by 64 columns. Whenever the
10 image portion requested has height or width which is not an even multiple of 64 the height or width of the image portion actually delivered by the server computer to the client computer will have to be padded out to an even multiple of 64 in order that the image portion constitutes an exact set of tiles. Hence more pixel information than the client requires is transmitted over the network in such a system.

The choice of the grid of uniformly-sized tiles also forces the system to transmit more header information than would otherwise be the case. Each image tile will have a header specifying such things as where it lies in the image, its resolution, the unique ID of the image to which belongs, etc. Consequently, if the pixel transmission packet consists of several tiles rather than a single pixel-block, there will also be
20 transmitted several tile headers instead of a single pixel-block header.

In summary, the choice of uniformly-sized tiles as the network transmission packet units is inefficient because of (a) padding: the transmission of extra pixels not actually needed for the client's current image display, and because of (b) excessive transmission of header information.

Thus it is desirable to provide a system and method that increases the efficiency of the network part of the client-server image-viewing system by abandoning the insistence of prior-art systems on transmitting uniformly-sized tiles over the network.

SUMMARY OF THE INVENTION

30 The network in a client-network-server imaging system is made more efficient by using non-uniform, optimally-sized image blocks in the network and client parts of the system. A uniform grid of tiles may still be used on the server part of the system, but is not required.

The image units requested by the client computer and transmitted by the server computer are non-uniformly-sized image blocks containing no unnecessary pixel information (thus eliminating the padding inefficiency), and with each of the non-uniformly-sized image blocks assigned the largest size practical given the client computer's imaging needs (thus mitigating the excessive transmission of header information problem). Given that the client computer is requesting non-uniformly-sized image blocks from the server, and is then receiving them back from the server, it is expeditious to have the client's cache of image data contain the exact same non-uniformly-sized image blocks received from the server.

The present invention uses optimally-sized non-uniform image blocks in the client and network parts of the client-network-server imaging system. The server computer's database of digital image may still contain pyramided, uniformly-tiled images, although neither the pyramiding nor the uniform-tiling of digital images is required by the present invention. In case the server computer's image database does contain uniformly-tiled images, the tiles in these images will differ in size and position from the custom-sized image blocks transmitted by the server to the client computer, and the server will have to perform a translation procedure to convert the former into the latter.

In summary, the present invention provides a method and system, including the transmission of non-uniform blocks from the server computer to the client computer, for rapid viewing of digital images.

There is thus provided in accordance with the preferred embodiment of the present invention a method for communicating a digital image over a network including storing a digital image on a server computer, maintaining on a client computer a cache of image blocks comprising portions of said digital image that have been downloaded from the server computer to the client computer, and, in response to a request by a user of the client computer for a given view, comprising a particular image portion at a given resolution, downloading from the server computer the optimal image portion required to render the view on the client computer.

The operations of the invention will be made clearer by the following detailed description of the preferred embodiments.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is an illustration of a system and method using non-uniform image blocks for rapid viewing of digital image files over a network in accordance with a preferred embodiment of the present invention;

5 Fig. 2 illustrates a sequence of instructions and operations performed by the system and method of Fig. 1;

Fig. 3 illustrates the detailed operation of the cache examination and image-block request steps of Fig. 2;

10 Fig. 4 illustrates the configurations that are possible in the partitioning step of Fig. 3;

Fig. 5 illustrates a view window into a larger image, showing the spatial organization of image blocks in the client computer's cache at the beginning of a viewing session;

15 Fig. 6 shows the same viewing session as in Fig. 5 after one panning operation subsequent to the first view; and

Fig. 7 shows the same viewing session as in Figures 5 and 2 after eight panning operations subsequent to the first view.

DESCRIPTION OF THE SPECIFIC EMBODIMENTS

20 Figure 1 illustrates the key components of a non-uniform image block client-server image-viewing system and method. Figure 2 illustrates a sequence of actions that may be performed under the systems and methods of Figure 1.

Server computer system 200 includes database 204 of digital images. Each digital image in database 204 exists in at least its primal-copy full-resolution form. This primal copy may or may not be partitioned into a uniform-tile grid, and it may or may not be supplemented by an image pyramid of downsampled lower-resolution images.

30 Client computer system 202 is connected to user-input devices 206, such as keyboard, mouse, microphone or trackball and to video display device 208. Client computer system 202 also includes main processor 210 which receives messages from input device 206 and communicates with other components of the client system.

A user of client computer system 202 uses user-input device 206 to send interactive image-viewing instruction 300 to client main processor 210. Image-viewing instruction 300 may be, for example, the instruction to display a rectangular subwindow

or view of a particular full-resolution image known to be resident in the digital image database 204 of server 200. The particular view to be displayed under instruction 300 will be identified by its resolution (relative to the full-resolution image) and by its coordinates (relative to a view of the entire image at the indicated resolution).

After receiving view instructions 300, the client main processor 210 performs cache examination task 302. Client computer system 202 includes image-block cache 212. This consists of a combination of writable electronic memory devices, such as computer random access memory (RAM) and hard disk units (HDUs) and of software to organize a list of image blocks that have already been received by client system 202 and are resident in client memory. Cache examination task 302 consists of subtasks 304 and 305 comprising (304) the identification of relevant image blocks resident in cache 212 (these are any image blocks in the cache which intersect the requisite view) and (305) the identification of the view residual. The view residual is the portion of the view that remains after all intersecting locally-resident blocks have been subtracted out at step 304.

Using methods that will be explained subsequently (with reference to Figures 3-7), client processor 210 partitions the residual area of the view (if there is any residual area) into one or more non-overlapping rectangles or image-blocks.

For each such residual image block, client main processor 210 then sends a request to the client system's message handling processor 214. The client message-handling processor 214 then issues image-block request 306 over network 216 to the message handler 218 of server 200. In request 306 the requested image block is identified by its resolution (relative to the full-resolution image) and by its coordinates (relative to a view of the entire image at the indicated resolution).

Server message handler 218 passes on the image-block request 306 to the server's image-block assembly processor 220. Image-block assembler 220 then examines the server's digital image database 204 in order to carry out image-block assembly operation 308, comprising reading out of database 204 a set of pixels necessary to construct the image block specified by message 306. If the resolution specified by message 306 is full-image resolution, or if it is the resolution of a subsampled image already present in an image pyramid in database 204, the requisite pixels will be obtainable from database 204 directly. Otherwise processor 220 will have to construct the requisite pixels by downsampling from the full-resolution image.

Once image-block assembler 220 has assembled the requisite block, it instructs server message handler 218 to execute image-block transmission step 310,

comprising transmitting the block over network 216 to client 202, where it is processed by client message handler 214. Client message handler 214 then informs client main processor 210 of the block's receipt, whereupon client main processor 210 executes step 312, comprising adding the image block to the client image-block cache.

5 In operations not illustrated in Figure 2, the client main processor 210 copies the image blocks required for the current view, cropped to the boundary of the current view, into the client computer system's display buffer 222, whereupon they are immediately visible in display 208. In various preferred embodiments of the present invention the requisite image blocks may be copied into display buffer 208 in various
10 orders. One embodiment does the following. After step 304, in which the locally-resident relevant image blocks are identified, each is copied into display buffer 222. Then, subsequently, as relevant image blocks are received from the server, they too are copied to display buffer 222, either before or after being added to the cache. Another embodiment is like the preceding with the additional feature that immediately after the
15 client issues request 306 to the server for a residual image block, it copies into display buffer 222 a temporary, imperfect stand-in for the missing image block, constructed by replicating pixels from the highest-resolution lower-resolution cached image blocks that intersect the residual area. When the actual image block required is later received, the operation of copying it into display buffer 222 overwrites and erases this temporary copy.

20 Figures 3 to 7 show in more detail the operations of the cache examination step 302 and image-block request step 306, in which image-blocks not-resident in the cache are identified and requested from the server.

Figure 3 is a flow-of-control diagram illustrating the operation of cache-examination step 302 and image-block request step 306. Figure 4 illustrates the
25 partitioning method used at step 406 of Figure 3. Figures 5 through 7 represent a view (102) into an image (100) and show the accumulation of image blocks in the client image-block cache as the view is panned over the image.

In a real panning operation, the rectangular view window remains fixed in position on display 208 and the display changes as if the larger image were moving under
30 the view window, with just a portion of the image viewable through the window. In Figures 5 through 7 the opposite is true. Representation 102 of the viewing window is moved over larger image 100 to show what portion of the image is visible at a given point in time. Cursor 104 in Figure 5, and analogous cursors in Figures 6 and 7, are artifices of the computer program used to generate Figures 5 through 7. In a real image-panning

interactive session, the direction and distance of each pan would most likely be controlled by keyboard arrow keys or by a mouse, with no display of a cursor like 104.

Initially, the user of client system 202 pans view window 102 so that it corresponds to the lower-central part of image 100 shown in Figure 5.

5 When the pan position is fixed, cache-examination process 302 begins as illustrated in Figure 3.

First, at step 400, client main processor 210 tries to find an image block in cache 212 that intersects view 102. Specifically, step 400 consists of (a) finding and identifying the first image block in cache 212 that intersects view 102, if there is any such
10 image block, end (b) reporting the absence of any such image block, if there is no block in the cache that intersects the view 102.

Next, at step 402, client main processor acts differentially depending on whether any such block was found.

The situation illustrated in Figure 5 is one in which the client image-block
15 cache is empty. Hence the report at step 402 will be that no intersecting block was found.

When no intersecting block is found at step 402, the client main processor then moves to step 410, at which it requests from the server a new block of image data filling the view. In Figure 5, the block requested at step 410 is block 0.

The client main processor then moves to step 412, at which point it quits
20 processing the current view. In the general situation, although the client main processor has quit processing the current view, it may still have other views to process, for the process of Figure 3 is potentially recursive, and there may be additional views awaiting processing at step 408. In the particular situation illustrated by Figure 5, there are no other views awaiting processing at step 408 after block 0 is requested, and the process
25 ends at step 412.

Consider now Figure 6, which shows a situation after block 0 has been received. Now view window 102 has panned to the upper right from its original position. Consider how the cache examination step 302 proceeds for the situation of Figure 6.

First, at step 400, the client main processor attempts to find a cached
30 image block intersecting view 102. It succeeds, finding block 0. Hence, at step 402, in response to the query, "Any block found?" the processor reports that yes, block 0 was found.

Next, at step 404, the client main processor asks whether any residual area remained in view 102 after it was intersected with block 0. The answer in this case is yes;

the residual area composed of rectangles 1 and 2 in Figure 6 remains after block 0 is subtracted away from view 102.

Next, at step 406, the client main processor partitions the residual area of the view into a minimum possible number of disjoint non-overlapping rectangular subviews. In the general case, when a first rectangle (the view) is overlapped by a second (the cached image block), and when the area of overlap is subtracted from the first rectangle, the residual area can then be minimally decomposed into between 1 and 4 rectangles. There are fifteen different intersection configurations, illustrated in Figure 4. The different configurations are characterized by the number and location of points of intersection between the cached image block and the view, and in some cases by additional information. It is apparent from the regularities in the fifteen configurations of Figure 4 that it is a straightforward task for a skilled computer programmer to program a computer to compute the coordinates of the residual rectangular subview or subviews remaining after intersection of the view with a cached rectangle.

In the case of the intersection of cached block 0 with view 102 in Figure 6, the minimum decomposition of the residual area is into two contiguous rectangular subviews, corresponding to not-yet-requested blocks 1 and 2. The case shown in Figure 6 corresponds to case (j) of Figure 4, with block 0 corresponding to block 440, and residual rectangular subviews 1 and 2 corresponding to subviews 441 and 442, respectively. Notice that there is some latitude in programming the decomposition. Case (j) of Fig. 4 has decomposed the residual area into a full-height segment 442 on the right and a less-than full-width segment 441 on the top. But it could just as well have been decomposed into a full-width segment on the top, and a less-than full-height segment on the top. A computer can be programmed to make such decomposition decisions arbitrarily and unvaryingly or algorithmically, so to achieve some goal, such as minimum variance in the sizes of the residual rectangles.

Then, at step 408, the client main processor iterates the process that begins at step 400 for each of the new views generated as subviews at step 406.

In the particular case illustrated in Figure 6, the client main processor first examines at step 400 the subview corresponding to block 1. It tries to find a cached image block intersecting this view. It finds no such intersecting block, whereupon it moves to step 410 and requests from the server a new block of image data, block 1, filling the view. Thereafter, the system moves to step 412, wherein it quits processing the view coinciding to block 1, and returns to step 408 to process any views remaining in the

queue. In this case, one unprocessed view remains at step 408, the view whose area corresponds to not-yet-requested block 2. The processor tries, and fails, at step 400, to find a cached image block intersecting the view whose area corresponds to not-yet-requested view 2. Hence it moves to step 410, whereupon it requests from the server new
5 block 2, filling the view. The processor then moves to step 412 whereupon it stops processing the view corresponding to view 2. The processor then asks whether any more views remain to be processed at step 408. The answer in this case is no, so the cache examination and request process illustrated in Figure 3 terminates.

It is well known in the art of computer programming to implement a
10 schema such as the one described above and illustrated in Figures 3 through 7 by a subroutine that calls itself recursively. The microfiche appended to the present patent document contains a set of subroutines (written in the C++ programming language to run on an Apple Macintosh computer) that carry out the sequence of operations described above by means of recursive calls to a procedure IntersectBox1, corresponding to step
15 400 of Figure 3.

At the end of the second pan operation, as shown in Figure 6, the client image block cache contains three cached blocks, one of which, block 0, is the full view size, and two of which, blocks 1 and 2, are fragments of the full view size.

Figure 7 shows the contents of the image-block cache after nine panning
20 operations. The third pan brings in block 3. The fourth pan brings in blocks 4, 5 and 6. The fifth pan brings in block 7. The sixth pan brings in block 8. The seventh pan brings in blocks 9, 10 and 11. The eighth pan brings in blocks 12, 13 and 14. The ninth pan brings in block 15.

Whenever a pan lands on an empty area, only one image block, comprising
25 exactly the pixels needed for the current view, is requested from the server and transmitted back to the client. Hence the amount of block-header messaging traffic from the server to the client is absolutely minimized. This is true of the pans that result in the requesting of blocks 0, 3, 7, 8 and 15.

When the area of a view intersects the boundary of the image, the image
30 block requested from the server is clamped to the edge of the image. This is true of block 8. The position of the image view, overlapping the image boundary, is shown by a dotted line around block 8 in Figure 7.

This completes the description of a first preferred embodiment of the present invention. A second preferred embodiment requires some discussion.

004270 9322950
09625226 072400

In the case of the first preferred embodiment, as illustrated in Figures 5 through 7, the set of pixels comprising the image blocks transmitted from the server to permit the display of a particular image view comprises exactly those pixels which are needed for the view but missing from the client image-block cache. No superfluous
5 pixels -- additional beyond those needed for the display of the current view -- are ever transmitted. There is no pixel padding, as there is in the case of prior-art uniform-tile client-server digital-image viewing systems.

However, in case the image resident on the server is compressed and it is desired to retain the transmitted image portions in their compressed form as they are
10 transmitted from the server to the client, it may be impossible to avoid some padding, and hence some transmission of superfluous pixels.

For example, it is common under Joint Photographic Experts Group (JPEG) compression schemes to compress images into 16 pixel by 16 pixel compressed squares, and each of these squares must be maintained intact between compression and
15 decompression in order for the compression-decompression process to work correctly.

A second embodiment of the invention is designed to work with JPEG-compressed image files resident on the server, or with images that have been compressed under other schemas that encode the image into a grid of uniformly-sized rectangles. In the second embodiment, the image is thought of as being uniformly tiled with a grid of
20 the size of the compression blocks. For example, if the compression unit is 16 rows by 16 columns, the tile grid would include pixel rows 0 through 15 in tile row 1, pixels 16 through 31 in tile row 2, and so forth. The actual view (102 in Figures 5 through 7), is now supplemented by a virtual view constructed as the smallest view-enveloping rectangle that corresponds exactly to a rectangular array of compression-grid tiles. Step
25 400 of Figure 3 is then decomposed into the following two substeps: (400a) Find the virtual view, defined as the minimum enveloping view corresponding exactly to a rectangular array of compression-grid tiles; (400b) try to find a cached image block intersecting the virtual view.

The modification described above will cause the dimensions of all image
30 blocks requested from the server to correspond exactly to a rectangular array of compression-grid tiles. Each such array is then transmitted in its compressed form from the server to the client. On receipt of each such array of compression-grid squares, the client decompresses them into an image-block which is then inserted into the client's image block cache 212. In this second preferred embodiment, the client may contain, in

addition to the image-block cache, a cache containing the compressed image blocks, as they were received from the server. The existence of this secondary cache gives the client an intermediate strategy at cache cleanup time, intermediate in severity between removing an image block from all client caches and retaining it in the principal image-block cache.

5 When the principal image-block cache hits its memory limit, the client may purge a set of image blocks from the principal cache while retaining the analogous compressed image blocks in the secondary compressed cache. In this manner, if the client finds it needs the decompressed image blocks again it can regenerate them from the compressed cache without having to go back to the server for them.

10 This completes the detailed description of preferred embodiments. It will be apparent to those skilled in the art, however, that the invention is not limited to these preferred embodiments, but includes other evident combinations, instantiations and extensions. In particular and for example, in the preferred embodiments described above, each of non-uniformly-sized rectangular image blocks requested by the client computer
15 from the server computer is assigned the largest size possible given the client computer's imaging needs. It may be desirable, for reasons of transmission and caching efficiency, to place an upper bound on the size of the image blocks requested by the client. The client then requests the largest image block possible subject to this upper bound. It may also be desirable, for reasons of transmission or caching efficiency, to insist that the image blocks
20 requested by the client have boundaries that correspond to byte boundaries or word boundaries or other particular boundaries in the server's digital memory. For this reason, the actual view (102 in Figures 4 through 6), may be supplemented by a virtual view constructed as the smallest view-enveloping rectangle with boundary positions exactly divisible by 8 or 16 or other number. The system's methodology for computing tile
25 requests is then analogous to the methodology it uses for requesting compression-grit tiles, as described above.

A number of schemas will be evident to those skilled in the art for increasing the speed of operation of step 400, which consists of trying to find an image block in the cache intersecting a given view. Such schemes include notably maintaining
30 the image blocks in linked lists according to the horizontal and vertical coordinates of their corners, and maintaining arrays of pointers from a regular grid of image cells, or from the rows and columns of the image, to all intersecting image blocks. In other variant embodiments, for example, the image blocks requested from the server and delivered to the client need not be restricted to rectangles, but may include more complex shapes. An

embodiment can be constructed in which the image blocks are any shape that may result from an agglomeration of rectangles. In this case, the amount of network traffic required to deliver the pixels required for a view is further reduced, since all pixels required for any view are sent in one packet with one header. In this embodiment, however, the

5 cache-examination step 402 comprising finding relevant cache-resident image blocks and identifying the image-block residual becomes computationally more difficult and hence more time consuming.

WHAT IS CLAIMED IS:

1 1. A method for communicating a digital image over a network,
2 comprising the steps of:
3 storing a digital image on a server computer;
4 maintaining on a client computer stored image blocks comprising portions
5 of said digital image that have been downloaded from the server computer to the client
6 computer;
7 in response to a request by a user of the client computer for a given view,
8 comprising a particular image portion at a given resolution, performing the steps of
9 computing the residual area of the view resulting from subtracting out
10 from the view the intersecting portions of stored image blocks, and, if the residual area is
11 positive,
12 downloading from the server computer the residual portion of the view at
13 the given resolution.

1 2. A method according to claim 1, further comprising partitioning the
2 residual portion of the view into the minimum possible number of rectangles and
3 downloading these rectangles from the server computer.

1 3. A method according to claim 2, wherein the operation of
2 partitioning the residual portion of the view into the minimum possible number of
3 rectangles comprises recursively executing, for the initial view and each subsequent
4 subview, a procedure that tries to find a first stored image-block intersecting the view,
5 and, if one is found, partitions the residual area of the view into the minimum possible
6 number of rectangular subviews, and, if none is found, treats the entire view as a
7 rectangle to be downloaded from the server computer.

1 4. A method according to claim 1, further comprising the step of
2 displaying the given view to the user.

1 5. A method according to claim 1, further comprising the step of
2 placing the downloaded image portion in the client computer's image-block cache.

1 6. A method according to claim 5, further comprising the step of
2 selectively removing items from the client computer's image-block cache when the size of
3 the cache reaches a limit.

1 7. A method for communicating a compressed digital image over a
2 network, the method comprising the steps of:

3 storing a compressed digital image, partitioned into a regular grid of
4 compression rectangles, on a server computer;

5 in response to a user request for a given view, comprising a particular
6 image portion at a given resolution,

7 constructing a virtual view, comprising the given view expanded as is
8 minimally necessary for its boundary to correspond to an integer number of whole
9 compression-grid tiles, and

10 ascertaining if any image blocks in the client computer's image-block
11 cache intersect the given virtual view, and then

12 computing the residual area of the virtual view resulting from subtracting
13 out from the virtual view the intersecting portions of cached image blocks, and, if the
14 residual area is positive,

15 downloading from the server computer a set of image compression-grid
16 cells comprising the residual portion of the virtual view at the given resolution.

1 8. A method according to claim 7, further comprising partitioning the
2 residual portion of the virtual view into the optimum number of rectangles corresponding
3 exactly to an integer number of whole compression-grid cells, and downloading these
4 rectangles from the server computer.

1 9. A method according to claim 8, wherein the operation of
2 partitioning the residual portion of the virtual view into the optimum possible number of
3 rectangles corresponding exactly to an integer number of whole compression-grid cells
4 comprises recursively executing, for the initial virtual view and each subsequent subview,
5 a procedure that tries to find a first cached image-block intersecting the view, and, if one
6 is found, partitions the residual area of the view into the minimum possible number of
7 rectangular subviews, and, if none is found, treats the entire view as a rectangle to be
8 downloaded from the server computer.

- 1 10. A method according to claim 7, further comprising the step of
2 displaying the given view to the user.
- 1 11. A method according to claim 7, further comprising the step of
2 decompressing the downloaded image portion and placing the decompressed image
3 portion in the client computer's image-block cache.
- 1 12. A method according to claim 7, further comprising the step of
2 placing the downloaded compressed image portion in the client computer's compressed
3 image-block cache.
- 1 13. A method according to claim 7, further comprising the step of
2 selectively removing items from the client computer's image-block cache when the size of
3 the cache reaches a limit.
- 1 14. A method according to claim 7, further comprising the step of
2 selectively removing items from the client computer's compressed image-block cache
3 when the size of the cache reaches a limit.
- 1 15. A method according to claim 7 comprising, if the residual area of
2 the view is positive, and before downloading from the server computer a set of image
3 compression-grid cells, ascertaining if any compressed image blocks from the
4 compressed image-block cache intersect the residual area, and, if so, decompressing them
5 and placing them in the image-block cache, and then constructing a new smaller residual
6 area of the view formed by subtracting from the residual area of the view the areas of the
7 image blocks that were just decompressed.
- 1 16. Apparatus for communicating a digital image over a network,
2 comprising:
3 a server computer including
4 a database of digital images;
5 a server message handler, operative to receive from client computers
6 requests for image blocks and to transmit image blocks to client computers;
7 a server image-block assembly processor, operative to examine the
8 server's digital image database in order to locate a specific digital image therein, and to

9 read out of the database of digital images a set of pixels necessary to construct a specific
10 image block; and
11 a client computer including
12 a cache of image blocks;
13 a client message handler, operative to send to the server computer requests
14 for image blocks and to receive said image blocks from the server computer; and
15 a client main processor, operative to receive from the client computer's
16 user a request for a particular view, comprising a particular image portion at a given
17 resolution, to ascertain if any image blocks in the client computer's image-block cache
18 intersect the given view, and to compute the residual area of the view resulting from
19 subtracting out from the view the intersecting portions of cached image blocks.

1 17. Apparatus according to claim 16, further comprising apparatus on
2 the client computer for partitioning the residual portion of the view into the minimum
3 possible number of rectangles.

1 18. Apparatus on the client computer according to claim 16 wherein
2 the operation of partitioning the residual portion of the view into the minimum possible
3 number of rectangles comprises recursively executing, for the initial view and each
4 subsequent subview, a procedure that tries to find a first cached image-block intersecting
5 the view, and, if one is found, partitions the residual area of the view into the minimum
6 possible number of rectangular subviews, and, if none is found, treats the entire view as a
7 rectangle to be downloaded from the server computer.

1 19. Apparatus according to claim 16, further comprising a display
2 device for displaying the given view to the user.

1 20. Apparatus according to claim 16, further comprising apparatus for
2 placing the downloaded image portion in the client computer's image-block cache.

1 21. Apparatus according to claim 16, further comprising apparatus for
2 selectively removing items from the client computer's image-block cache when the size of
3 the cache reaches a limit.

1 22. Apparatus according to claim 16 wherein some of the digital
2 images in the server image database are compressed into regular grids of compression

004220 3252960

3 rectangles,, wherein the server image-block assembly processor is operative to examine
4 the server's digital image database in order to locate a specific compressed digital image
5 therein, and to read out of the database of digital images a set of compression grid cells
6 necessary to construct a specific compressed image block.

1 23. Apparatus according to claim 16 wherein the client main processor
2 is operative to receive from the client computer's user a request for a particular view,
3 comprising a particular image portion at a given resolution, to construct a virtual view,
4 comprising the given view expanded as is minimally necessary for its boundary to
5 correspond to an integer number of whole compression-grid tiles, to ascertain if any
6 image blocks in the client computer's image-block cache intersect the given virtual view,
7 and to compute the residual area of the virtual view resulting from subtracting out from
8 the virtual view the intersecting portions of cached image blocks.

1 24. Apparatus according to claim 16, further comprising, on the client
2 computer, a cache of compressed image blocks.

1 25. Apparatus according to claim 16, further comprising apparatus for
2 decompressing the downloaded image portion and placing the decompressed image
3 portion in the client computer's image-block cache.

1 26. Apparatus according to claim 16, further comprising apparatus for
2 placing the downloaded compressed image portion in the client computer's compressed
3 image-block cache.

1 27. Apparatus according to claim 16, further comprising apparatus for
2 selectively removing items from the client computer's image-block cache when the size of
3 the cache reaches a limit.

1 28. Apparatus according to claim 16, further comprising apparatus for
2 selectively removing items from the client computer's compressed image-block cache
3 when the size of the cache reaches a limit.

1 29. Apparatus according to claim 16 comprising apparatus operative, if
2 the residual area of the view is positive, and before downloading from the server
3 computer a set of image compression-grid cells, for ascertaining if any compressed image

- 4 blocks from the compressed image-block cache intersect the residual area, and, if so,
- 5 decompressing them and placing them in the image-block cache, and then constructing a
- 6 new smaller residual area of the view formed by subtracting from the residual area of the
- 7 view the areas of the image blocks that were just decompressed.

METHOD AND SYSTEM USING NON-UNIFORM IMAGE BLOCKS FOR RAPID INTERACTIVE VIEWING OF DIGITAL IMAGES OVER A NETWORK

ABSTRACT OF THE DISCLOSURE

5 Apparatus and method for rapid interactive viewing of a digital image over a
network. A client computer displays a view of an image. The image is originally resident on
a server computer. The client computer maintains a cache, initially empty, of image blocks
already obtained from the server. When the client computer is asked to render a view of a
particular portion of the image at a particular resolution, it first ascertains if any image blocks
10 in the image-block cache intersect the requested view. It then computes the residual area of
the view resulting from subtracting out from the view the intersecting portions of cached
image blocks, and, if the residual area is positive, downloads from the server computer a set
of image blocks comprising the residual portion of the view at the given resolution.

15
SF 1111886 v2


```
graph TD
    subgraph Server [ ]
        DIB[Digital Image Database]
        IBAP[Image-Block Assembly Processor]
        SMH[Server Message Handler]
        DIB <--> IBAP
        IBAP <--> SMH
    end
    subgraph Client [ ]
        CMH[Client Message Handler]
        CMP[Client Main Processor]
        IBC[Image-Block Cache]
        DB[Display Buffer]
        CMH <--> CMP
        IBC <--> CMP
        CMP --> DB
    end
    UserInput[User-Input Devices] --> CMP
    Display[Display]
    Network[Network]

    204 --> DIB
    220 --> IBAP
    200 --> SMH
    218 --> SMH
    SMH -- 216 --> Network
    Network --> CMH
    214 --> CMH
    202 --> Client
    UserInput --> CMP
    212 --> IBC
    210 --> CMP
    222 --> DB
    DB --> Display
    208 --> Display
    206 --> UserInput
```

The diagram illustrates a network-based image processing system. It consists of a server side and a client side connected via a Network. The server side includes a Digital Image Database (204), an Image-Block Assembly Processor (220), and a Server Message Handler (200, 218). The client side includes a Client Message Handler (214), a Client Main Processor (210), an Image-Block Cache (212), and a Display Buffer (222). User-Input Devices (206) are connected to the Client Main Processor. The Display (208) is connected to the Display Buffer. The Network (216) connects the Server Message Handler to the Client Message Handler.

Figure 1

```
graph TD; 300 --> A[Interactive Image-Viewing Instruction]; A --> B[Identify relevant cache-resident image blocks]; 304 --> B; B --> C[Image-block residual identification]; 305 --> C; C --> D[Image-block request]; 306 --> D; D --> E[Image-block assembly]; 308 --> E; E --> F[Image-block transmission]; 310 --> F; F --> G[Add image block to cache]; 312 --> G; subgraph "Cache Examination"; B; C; end
```

The flowchart illustrates the process of adding an image block to a cache. It begins with an 'Interactive Image-Viewing Instruction' (300), which leads to a 'Cache Examination' block. Inside this block, the process involves 'Identify relevant cache-resident image blocks' (304) and 'Image-block residual identification' (305). The 'Cache Examination' block then leads to an 'Image-block request' (306), which leads to 'Image-block assembly' (308). The 'Image-block assembly' block leads to 'Image-block transmission' (310), which finally leads to 'Add image block to cache' (312).

Figure 2

004240" 92252960

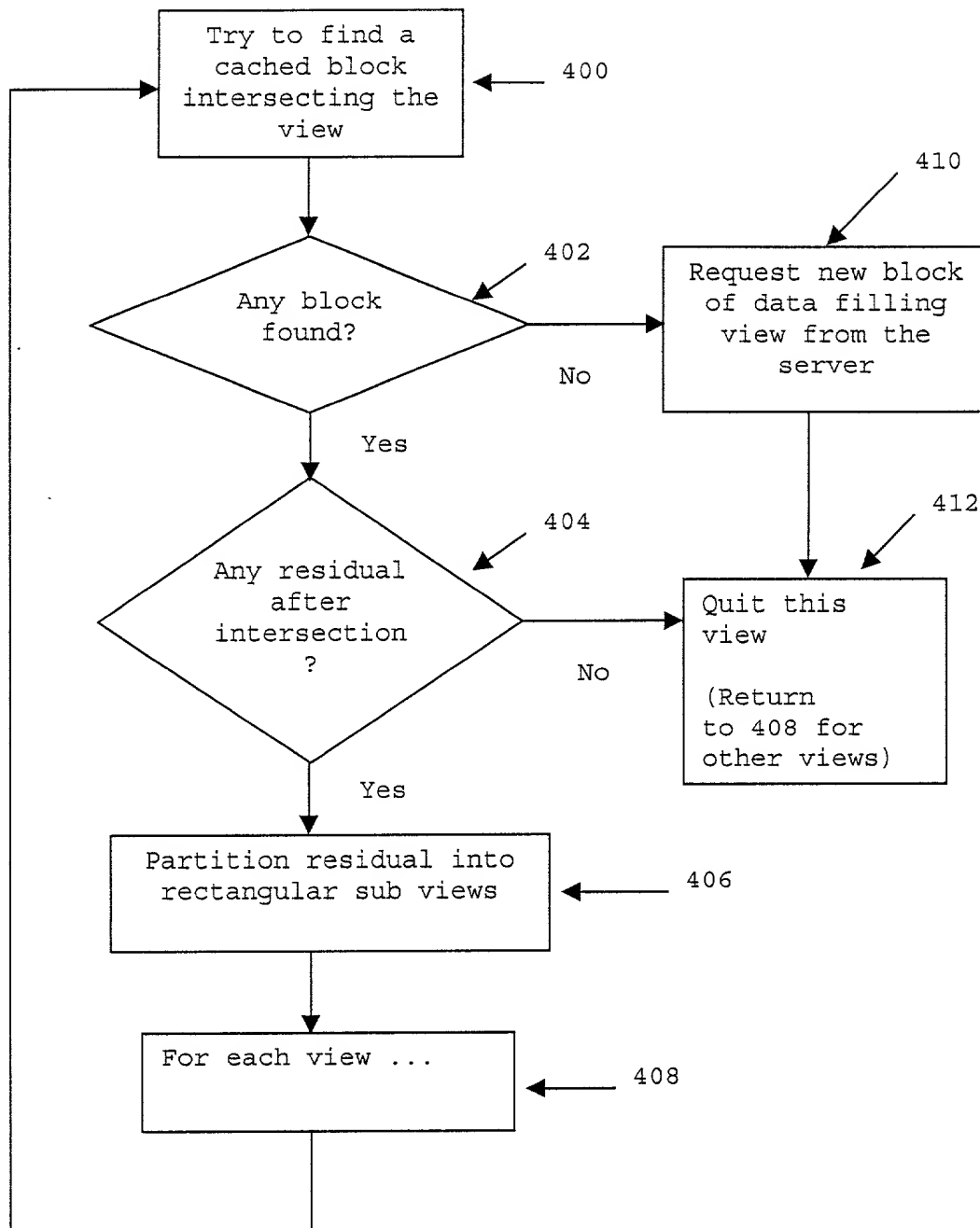


Figure 3

004260" 92252960

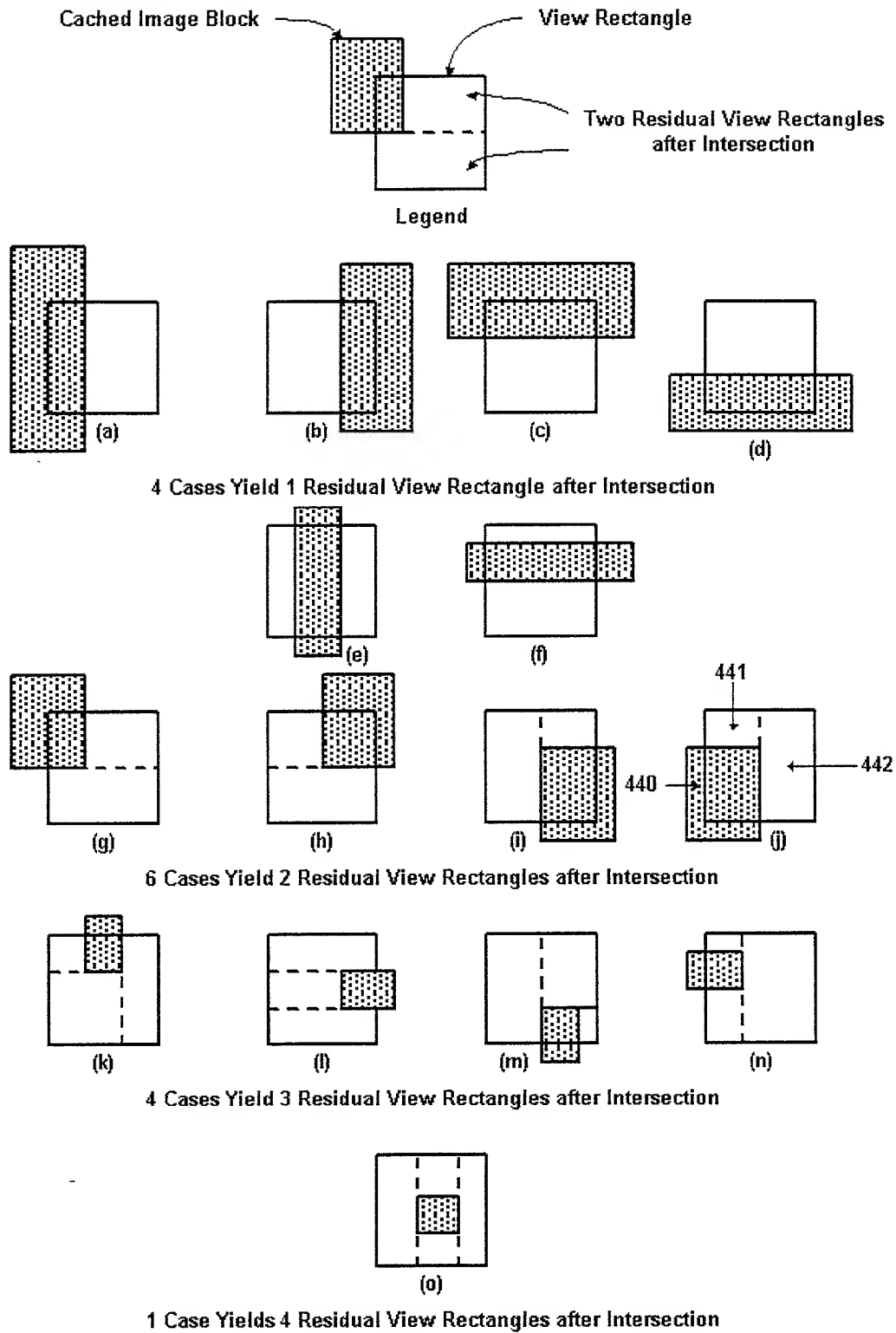


Fig. 4

004240" 92252960

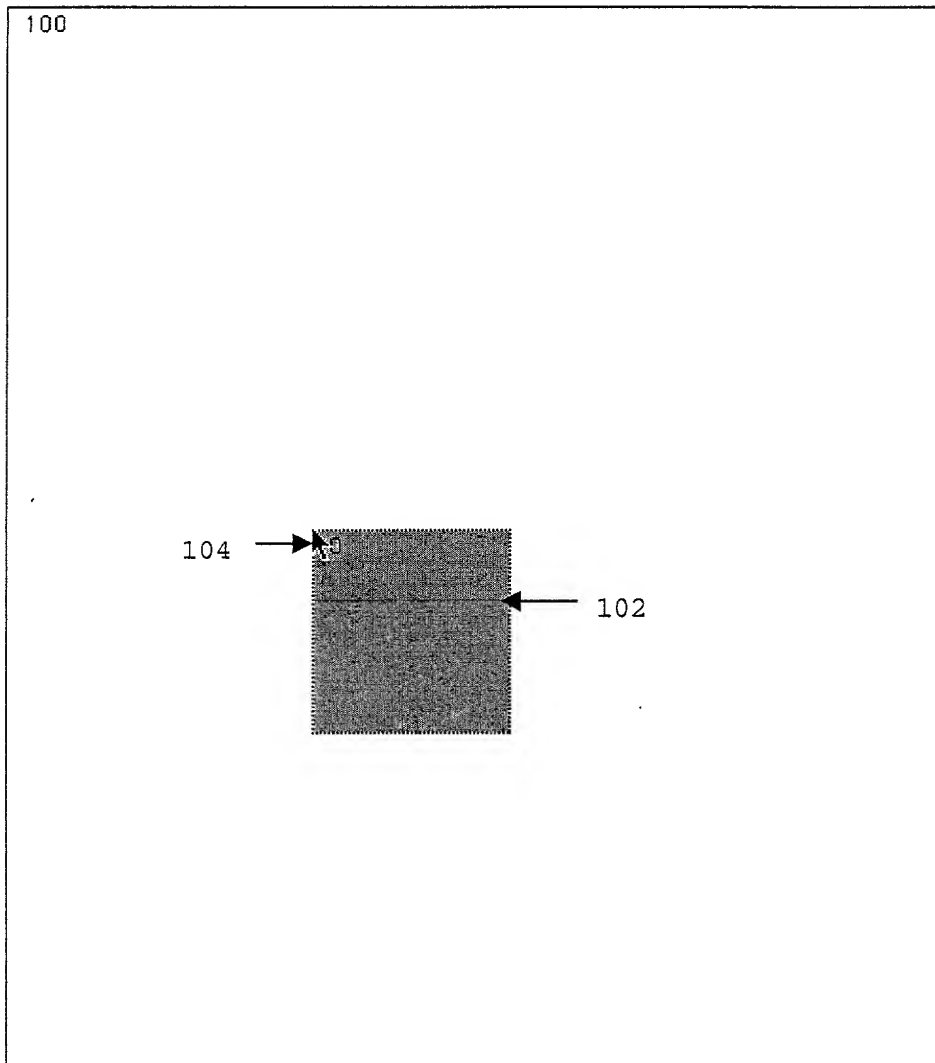


Figure 5.

004640" 9222960

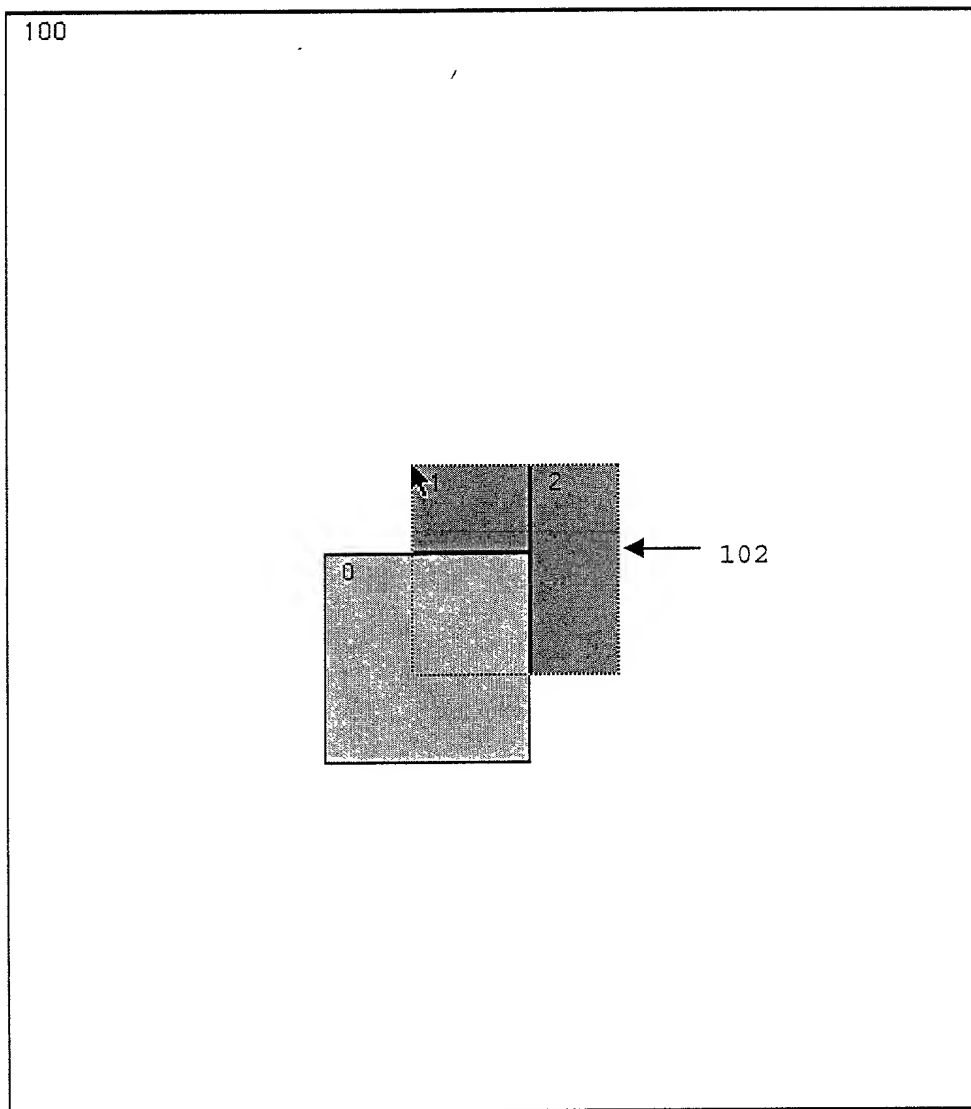


Figure 6

00420" 92292960

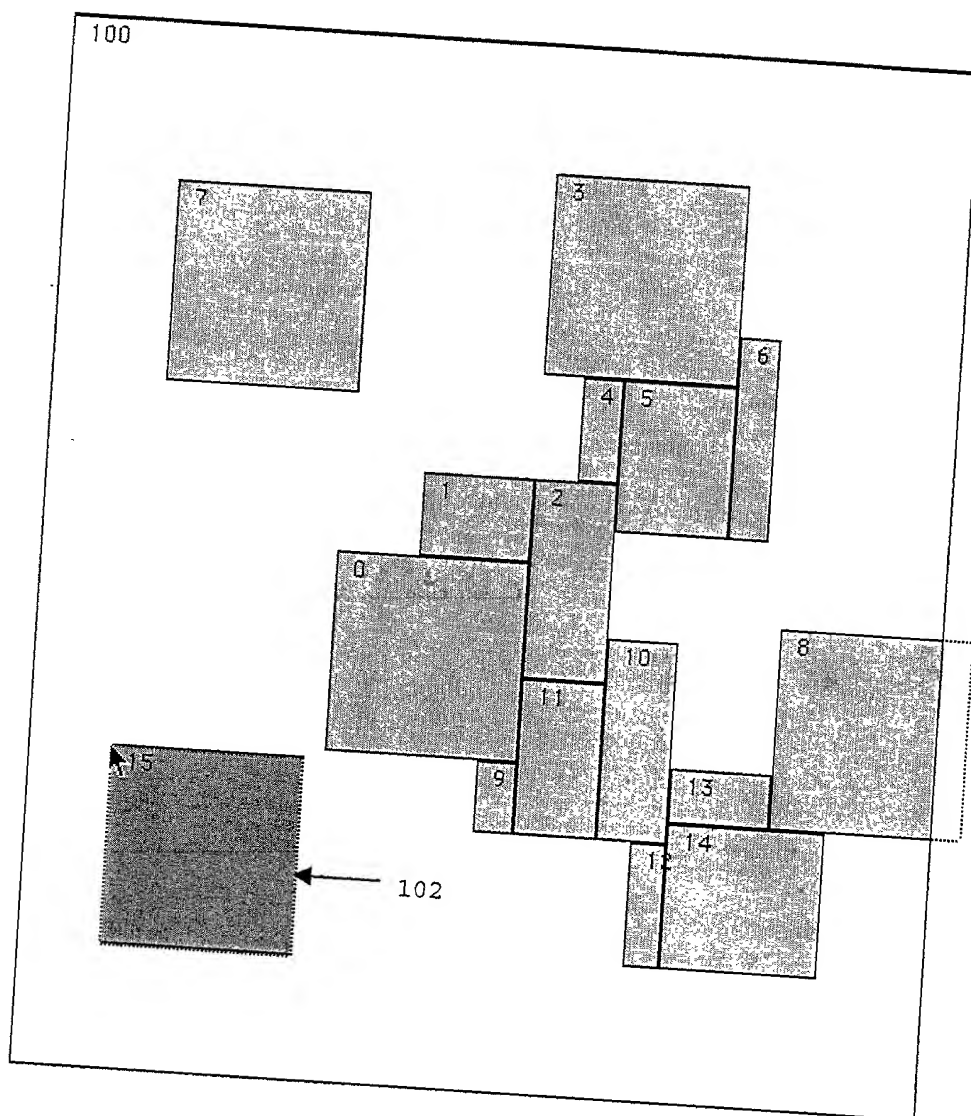


Figure 7

DECLARATION

As a below named inventor, I declare that:

My residence, post office address and citizenship are as stated below next to my name; I believe I am the original, first and sole inventor (if only one name is listed below) or an original, first and joint inventor (if plural inventors are named below) of the subject matter which is claimed and for which a patent is sought on the invention entitled: **METHOD AND SYSTEM USING NON-UNIFORM IMAGE BLOCKS FOR RAPID INTERACTIVE VIEWING OF DIGITAL IMAGES OVER A NETWORK** the specification of which X is attached hereto or was filed on as Application No. and was amended on (if applicable).

I have reviewed and understand the contents of the above identified specification, including the claims, as amended by any amendment referred to above. I acknowledge the duty to disclose information which is material to patentability as defined in Title 37, Code of Federal Regulations, Section 1.56. I claim foreign priority benefits under Title 35, United States Code, Section 119 of any foreign application(s) for patent or inventor's certificate listed below and have also identified below any foreign application for patent or inventor's certificate having a filing date before that of the application on which priority is claimed.

Prior Foreign Application(s)

Country	Application No.	Date of Filing	Priority Claimed Under 35 USC 119

I hereby claim the benefit under Title 35, United States Code § 119(e) of any United States provisional application(s) listed below:

Application No.	Filing Date

I claim the benefit under Title 35, United States Code, Section 120 of any United States application(s) listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States application in the manner provided by the first paragraph of Title 35, United States Code, Section 112, I acknowledge the duty to disclose material information as defined in Title 37, Code of Federal Regulations, Section 1.56 which occurred between the filing date of the prior application and the national or PCT international filing date of this application:

Application No.	Date of Filing	Status

Full Name of Inventor 1:	Last Name: WENSLEY	First Name: PAUL	Middle Name or Initial:	
Residence & Citizenship:	City: Sausalito	State/Foreign Country: California	Country of Citizenship: United States	
Post Office Address:	Post Office Address: 90 George Lane	City: Sausalito	State/Country: California	Postal Code: 94965
Full Name of Inventor 2:	Last Name: MINNER	First Name: RICHARD	Middle Name or Initial: T.	
Residence & Citizenship:	City: Carmichael	State/Foreign Country: California	Country of Citizenship: United States	
Post Office Address:	Post Office Address: 2635 Napoli Court	City: Carmichael	State/Country: California	Postal Code: 95608

I further declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of the application or any patent issuing thereon.

Signature of Inventor 1	Signature of Inventor 2
<hr/>	<hr/>
Paul Wensley	Richard T. Minner
Date	Date

SF 1118434 v1


```

// =====
// File: GenCookie.c
// Copyright, 1999, 2000, Xippix, Inc.
// The terminology "cookie" is because the pattern of image blocks selected
// looks, when graphed, like the pattern of a rectangular cookie cutter
// stamped in an overlapping manner on dough.)
// =====
// constants from TPCApp.h
#include <CMAApplication.h>

// =====
// Resource-dependent constants
// If you port these to another application, you have to make sure all these
// are set up in the resource file. E.g., set up window 1450 (WIND_Tonal)
// for the tonal budgeting window.
// =====
// constants from TPCApp.h
#include "LApplication.h" // You have to call this before you declare const.

#include "FontRs.h"
const ResIDT WIND_Tonal      = 1450; // TpcRs_h
const ResIDT WIND_picker    = 10800; // ViewMsg_h
const ResIDT WIND_pickerS   = 10801;
const ResIDT WIND_Block     = 11100; // TpcRs_h
#include "GeneralData.h"
#include "GenPreview.h"
#include "GenPreview.cmd" // Defines of messages from GenPreview, form
"msg_{foobar}"
#include "HLS.h"
#include "ViewBigCube.h"

#include "NumScrollBar.h"
#include "DblScrollBar.h"

#include "GenCookie.h"
#include "Tonal.h"

#include "JBfileUtils.h"
#include "busy.h"
// CW10 Version had: #include <String_Utils.h> // for c2pstr

#include "tpu_pane.h"
#include "tpu_alert.h"
#include "tpu_number.h"
#include "tpu_trig.h"

#include "AGA.h"
#include "AGAPP.h"

#include <Quickdraw.h>

#include <math.h> // for "pow"

// TL file system
#include "TLTypeDefs.h" // Has typedefs "s8", etc., needed by TLFileSys.h
#include "TLFileSys.h"

```

```

// Following are for the interface to the application data
#include <stdlib.h>

static int hasdata = 0;

#ifdef MEM_DEBUG
#include "smrtheap.hpp"
// #include "shmac.c"
#endif

static int staticbreak = 0;
static unsigned char staticchar = 0;
static unsigned char flipping = 1;

// Block Stuff
// Following leftover from TPCViewPreview.c
static int wwidth = 510; // use 0 to 499 incl. 320; // 460
static int wheight = 570; // use 0 to 559 incl. 460;
static int finalx = 449;
static int finaly = 499;
static int panw = 96; // 12 * 8bits;
static int panh = 96; // 96 rows thereof
static unsigned char DoDrawBlock = 0;

static unsigned char everdrawn = 0;
static unsigned char newway = 1;

static void ToggleDoDraw()
{
    if (!DoDrawBlock)
        DoDrawBlock = 1;
    else
        DoDrawBlock = 0;
}

static RGBColor gray = {0xc000, 0xc000, 0xc000};
static RGBColor dark = {0x9000, 0x9000, 0x9000};
static RGBColor black = {0x0000, 0x0000, 0x0000};
static RGBColor white = {0xffff, 0xffff, 0xffff};

// GenCookie.h
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/
//
// Start class ViewCookie
//
// Cooked
/* Proc */ ViewCookie::ViewCookie
(
    const SPaneInfo &tpi,
    const SViewInfo &tvi,
    unsigned short yoffset, // GenTonal.h
    WinCookie *dlgbox,

```

00720"072400

004240 322950

```

                                LCommander *inSuperCommander,
                                GeneralData *gen
                                )
                                : LView (tpi, tvi)
{
    capp__ = inSuperCommander;          // Untyped ptr to App.
    gen__ = gen;
    dlg__ = dlgbox;

    secondPort = 0;

    Rect frame;
    CalcLocalFrameRect(frame);
    gframe = frame; // Block;

    account = lcount = 0; // Block;

    stackcount = 0;

    image.tl.x = 0;
    image.tl.y = 0;
    image.br.x = finalx;
    image.br.y = finaly;
    image.id = 100;
    image.status = 0; // 0 if no image data for the block; 1 if here; 2 if on
order    image.next = 0; // we don't care. It's a singleton.

    first = -1;

    blonkall.left = 0;
    blonkall.right = finalx + 1; // PaintRect covers its top & left parms,
    blonkall.top = 0;
    blonkall.bottom = finaly + 1;

    // Next action at ClickSelf; then AdjustCursorSelf
}

// Cooked
unsigned char ViewCookie::DoBlocksIntersect (uRect *a, uRect *b, unsigned char
DoCalc)
{
    if ( (a->tl.x <= b->br.x) && (b->tl.x <= a->br.x) &&
        (a->tl.y <= b->br.y) && (b->tl.y <= a->br.y) )
    {
        if (DoCalc)
        {
            if (a->tl.x > b->tl.x) intercept.tl.x = a->tl.x; else
intercept.tl.x = b->tl.x;
            if (a->tl.y > b->tl.y) intercept.tl.y = a->tl.y; else
intercept.tl.y = b->tl.y;
            if (a->br.x < b->br.x) intercept.br.x = a->br.x; else
intercept.br.x = b->br.x;
            if (a->br.y < b->br.y) intercept.br.y = a->br.y; else
intercept.br.y = b->br.y;
        }
    }
}
```

```

        return (1);
    }
    else
        return (0);
}

// Find first intersecting block.
// Cooked
int ViewCookie::FindBlockIntersectingBlock (uRect *target)
{
    int i; // Block
    unsigned char finished = 0;
    i = first;

    if (first < 0)
        return (-1); // Failure

    // 0-terminated linked list.
    while (!finished)
    {
        if (DoBlocksIntersect (target, &rlist[i], 1))
            return (i);
        i = rlist[i].next;
        if (i < 0)
            finished = 1;
    }
    return (-1); // Failure.
}

void ViewCookie::ReceiveOrderedBoxes ()
{
    int i; // Block
    unsigned char finished = 0;
    i = first;

    // 0-terminated linked list.
    while (!finished)
    {
        if (rlist[i].status == 2)
            rlist[i].status = 1;
        i = rlist[i].next;
        if (i < 0)
            finished = 1;
    }
}

int ViewCookie::FindPredecessor (int block)
{
    int i, j; // Block
    unsigned char finished = 0;
    i = first;

    // 0-terminated linked list.
    while (!finished)

```

00420"3222360

```

    {
        j = rlist[i].next;
        if (j == block)
            return (i);
        i = j;
        if (j < 0)
            finished = 1;
    }
    return (-1); // Failure.
}

```

```

void ViewCookie::DelRectangle (int block)
{
    int predecessor;

    if (block != first)
        // Stitch linked list back together
        {
            predecessor = FindPredecessor (block);
            rlist[predecessor].next = rlist[block].next;
        }
    else
        // if (block == first)
        first = rlist[block].next;

    lcount--;
}

```

```

// GenBlock.h
void ViewCookie::AddRectangle (int minx, int miny, int maxx, int maxy, int
status)
{
    // Set rectangle 1
    rlist[acount].tl.x = minx;
    rlist[acount].tl.y = miny;
    rlist[acount].br.x = maxx;
    rlist[acount].br.y = maxy;
    rlist[acount].id = acount;
    rlist[acount].status = status;
    rlist[acount].next = first; // Temp. Add as new first in list. The list is
(-1)-terminated.

    first = acount; // Temp.
    acount++;
    lcount++;
    // SewInLists(); // Later: Keep four linked list of next pointers in the
order of:
    // tl.x, tl.y, br.x, br.y. This will speed up searches.
    (FindBlockIntersectingBlock, etc.)
}

```

```

static unsigned char PutABreakpointHere()
{
    staticbreak++;
}

```

00420-02252960

```

    return (staticbreak);
}

```

```

static unsigned char PutABreakpointThere()
{
    unsigned char fazz = PutABreakpointHere();
    return (fazz);
}

```

```

// Cooked
unsigned char ViewCookie::ItsTall (uRect *r)
{
    if ((r->br.y - r->tl.y) > (r->br.x - r->tl.x))
        return (true);
    else
        return (false);
}

```

```

// Cooked
void ViewCookie::FactorBlock(uRect *box)
{
    int i, j;
    int countincluded;
    unsigned char tlin, trln, blin, brin; // top left in, etc
    unsigned char ht, hb, vl, vr; // horizontal top, etc. for 0 intercepts
case. unsigned char tl, bl, tr, br; // top left, etc. for 1 intercept
case. unsigned char ls, rs, ts, bs; // left side, etc. for 2 intercept
case. unsigned char fazz; // Debug.

    if (!acount)
        fazz = PutABreakpointHere();

    int minxi, maxx, minyi, maxyi;

    int blockminxi = box->tl.x; // Break these out for easier parsing.
    int blockmaxxi = box->br.x;
    int blockminyi = box->tl.y;
    int blockmaxyi = box->br.y;

    minxi = intercept.tl.x; // Break these out for easier parsing.
    maxx = intercept.br.x;
    minyi = intercept.tl.y;
    maxyi = intercept.br.y;

    // To HERE

    // Get the count of included points.
    countincluded = 0;
    tlin = trln = blin = brin = 0;

```

004240 0222200


```

/*
WARNING!! It took me a long time to get the degree-of-inclusiveness
(the choice of "<" vs "<=") right in the following. The solution turns
out to be that a point is in if it leaves white space on the continuation
of both of its joining edges. E.g., the bottom right point is in if it
leaves white space at the bottom (maxyi < blockmaxyi) and at the right
(maxxi < blockmaxxi). As far as the remaining two inequalities goes,
they can be satisfied weakly: (maxxi >= blockminxi; maxyi >= blockminyi).
*/

// Top left
// Must leave white space at left (blockminxi) and top (blockminyi).
if (minxi > blockminxi && minxi <= blockmaxxi &&
    minyi > blockminyi && minyi <= blockmaxyi) // WARNING!! See note above
{
    countincluded++;
    tlin = 1;
}

// Top right
// Must leave white space at right (blockmaxxi) and top (blockminyi).
if (maxxi >= blockminxi && maxxi < blockmaxxi &&
    minyi > blockminyi && minyi <= blockmaxyi) // WARNING!! See note
above
{
    countincluded++;
    trin = 1;
}

// Bottom left
// Must leave white space at left (blockminxi) and bottom (blockmaxyi).
if (minxi > blockminxi && minxi <= blockmaxxi &&
    maxyi >= blockminyi && maxyi < blockmaxyi) // WARNING!! See note
above
{
    countincluded++;
    blin = 1;
}

// Bottom right
// Must leave white space at right (blockmaxxi) and bottom (blockmaxyi).
if (maxxi >= blockminxi && maxxi < blockmaxxi &&
    maxyi >= blockminyi && maxyi < blockmaxyi) // WARNING!! See note above
{
    countincluded++;
    brin = 1;
}

switch (countincluded)
{

```

004240 03252960

```

default:
case 0:
    // We will make 2 new ones or 3 new ones
    // Watch out for the difference between "<" and "<="; the
    // question is always: Is white space left.
    vl = vr = ht = hb = 0;
    if (maxxi < blockmaxxi && maxxi >= blockminxi)
        vl = 1;
    if (minxi > blockminxi && minxi <= blockmaxxi)
        vr = 1;
    if (maxyi < blockmaxyi && maxyi >= blockminyi)
        ht = 1;
    if (minyi > blockminyi && minyi <= blockmaxyi)
        hb = 1;

    //////////////////////////////////////
/
    //
    // First vertical-region cases
    if (vl && !vr)
    {
        //
        // -----
        // |.1.| 2 |
        // |...|   |
        // |...|   |
        // -----
        //
        // Region 1 is the view
        IntersectBox4 (maxxi+1,    blockminyi, blockmaxxi,
blockmaxyi); // Rect 2
    }
    else
    if (vr && !vl)
    {
        //
        // -----
        // | 1 |.2.....|
        // |   |.....|
        // |   |.....|
        // -----
        //
        // Region 2 is the view
        IntersectBox4 (blockminxi, blockminyi, minxi-1,
blockmaxyi); // Rect 1
    }
    else
    if (vr && vl)
    {
        //
        // -----
        // | 1 |.2.....|3 |
        // |   |.....|   |
        // |   |.....|   |
        // -----
        //
        // Region 2 is the view
        IntersectBox4 (blockminxi, blockminyi, minxi-1,
blockmaxyi); // Rect 1
        IntersectBox4 (maxxi+1,    blockminyi, blockmaxxi,
blockmaxyi); // Rect 3
    }
}

```

```

////////////////////////////////////
/
    //
    // Then horizontal-region cases
    else
    if (ht && !hb)
    {
    //
    // -----
    // |.1.....|
    // |-----| Region 1 is the view
    // | 2      |
    // |-----|
    IntersectBox4 (blockminxi, maxyi+1,    blockmaxxi,
blockmaxyi); // Rect 2
    }
    else
    if (hb && !ht)
    {
    //
    // -----
    // | 1      |
    // |-----| Region 2 is the view
    // |.2.....|
    // |-----|
    IntersectBox4 (blockminxi, blockminyi, blockmaxxi, minyi-1);

// Rect 1
    }
    else
    if (hb && ht)
    {
    //
    // -----
    // | 1      |
    // |-----| Region 2 is the view
    // |.2.....|
    // |-----|
    // | 3      |
    // |-----|
    IntersectBox4 (blockminxi, blockminyi, blockmaxxi, minyi-1);

// Rect 1
    IntersectBox4 (blockminxi, maxyi+1,    blockmaxxi,
blockmaxyi); // Rect 3
    }
    break;

case 1:
    // We will make 3 new ones
    tl = tr = bl = br = 0;
    if (maxxi < blockmaxxi && maxyi < blockmaxyi)
        tl = 1;
    else
    if (maxxi < blockmaxxi && minyi > blockminyi)
        bl = 1;
    else
    if (minxi > blockminxi && maxyi < blockmaxyi)

```



```

// |-----|
// |.2.....| 3 |
// |.....|
// |.....|
// |-----|
//
// IntersectBox4 (blockminxi, blockminyi, blockmaxxi, minyi-1);
// Rect 1
// IntersectBox4 (maxxi+1, minyi, blockmaxxi,
blockmaxyi); // Rect 3
// } // if (bl) tall
// else
// {
// // Wide
// // -----
// // | 1 | 3 |
// // | | |
// // | | |
// // |-----|
// // |.2.....|
// // |.....|
// // |-----|
// //
// IntersectBox4 (blockminxi, blockminyi, maxxi, minyi-1);
// Rect 1
// IntersectBox4 (maxxi+1, blockminyi, blockmaxxi,
blockmaxyi); // Rect 3
// } // if (bl) wide
// } // if (bl)
//
// if (tr)
// {
// if (!flipping || ItsTall(box))
// {
// // Tall
// // -----
// // | 1 |.2...|
// // | |.....|
// // | |.....|
// // | |.....|
// // |-----|
// // | 3 |
// // | |
// // | |
// // |-----|
// //
// IntersectBox4 (blockminxi, blockminyi, minxi-1, maxyi);
// Rect 1
// IntersectBox4 (blockminxi, maxyi+1, blockmaxxi,
blockmaxyi); // Rect 3
// } // if (tr) tall
// else
// {
// // Wide

```

004032360

```

// -----
// | 1          |.2...|
// |          |.....|
// |          |.....|
// |          |.....|
// |          |-----|
// |          | 3    |
// |          |-----|
// -----
//
IntersectBox4 (blockminxi, blockminyi, minxi-1,
blockmaxyi); // Rect 1
IntersectBox4 (minxi,          maxyi+1,    blockmaxxi,
blockmaxyi); // Rect 3
} // if (tr) wide
} // if (tr)

if (br)
{
if (!flipping || ItsTall(box))
{
// Tall
// -----
// | 1          |
// |          |
// |          |
// |          |
// |-----|
// | 2          |.3...|
// |          |.....|
// |          |.....|
// |          |.....|
// |-----|
// -----
//
IntersectBox4 (blockminxi, blockminyi, blockmaxxi, minyi-1);
// Rect 1
IntersectBox4 (minxi,          minyi,      blockmaxxi,
blockmaxyi); // Rect 3
} // if (tr) tall
else
{
// Wide
// -----
// | 1          | 2    |
// |          |
// |          |
// |          |
// |-----|
// |          |.3...|
// |          |.....|
// |-----|
// -----
//
IntersectBox4 (blockminxi, blockminyi, minxi-1,
blockmaxyi); // Rect 1
IntersectBox4 (minxi,          blockminyi, blockmaxxi, minyi-1);
// Rect 2
} // if (tr) wide

```

004220 92252960

```

    } // if (tr)

    break;

case 2:
    // We will make 4 new ones
    ls = rs = ts = bs = 0;
    if (tlin && blin)
        rs = 1;
    else
        if (trlin && brin)
            ls = 1;
        else
            if (tlin && trlin)
                bs = 1;
            else
                if (blin && brin)
                    ts = 1;

    if (ts)
    {
        if (!flipping || ItsTall(box))
        {
            // Tall
            //
            // -----
            // | 1 | .2..... | 3 |
            // |   | .....   |   |
            // |-----|
            // | 4 |
            // |
            // |
            // |
            // |
            // |-----|
            //
            IntersectBox4 (blockminxi, blockminyi, minxi-1,    maxyi);
            // Rect 1
            IntersectBox4 (maxxi+1,    blockminyi, blockmaxxi, maxyi);
            // Rect 3
            IntersectBox4 (blockminxi, maxyi+1,    blockmaxxi,
blockmaxyi); // Rect 4
        } // End if (ts) tall
        else
        {
            // Wide
            //
            // -----
            // | 1 | .2..... | 3 |
            // |   | .....   |   |
            // |   |-----|
            // |   | 4 |
            // |   |
            // |   |
            // |   |
            // |-----|
            //
            //

```

```

        IntersectBox4 (blockminxi, blockminyi, minxi-1,
blockmaxyi); // Rect 1
        IntersectBox4 (maxxi+1,    blockminyi, blockmaxxi,
blockmaxyi); // Rect 3
        IntersectBox4 (minxi,      maxyi+1,    maxxi,
blockmaxyi); // Rect 4
    } // End if (ts) wide
    } // End if (ts)

    if (bs)
    {
        if (!flipping || ItsTall(box))
        {
            // Tall
            //
            // -----
            // | 1
            // |
            // |
            // |
            // |
            // |
            // |-----|
            // | 2 |.3.....| 4
            // | |.....|
            // |-----|
            IntersectBox4 (blockminxi, blockminyi, blockmaxxi, minyi-1);
            // Rect 1
            IntersectBox4 (blockminxi, minyi,      minxi-1,    maxyi);
            // Rect 2
            IntersectBox4 (maxxi+1,    minyi,      blockmaxxi, maxyi);
            // Rect 4
        } // End if (bs) tall
        else
        {
            // Wide
            //
            // -----
            // | 1 | 2 | 3
            // |
            // |-----|
            // | |.4.....|
            // | |.....|
            // | |.....|
            // |-----|
            //
            IntersectBox4 (blockminxi, blockminyi, minxi-1,
blockmaxyi); // Rect 1
            IntersectBox4 (minxi,      blockminyi, maxxi,      minyi-1);
            // Rect 2
            IntersectBox4 (maxxi+1,    blockminyi, blockmaxxi,
blockmaxyi); // Rect 3
        } // End if (bs) wide
    } // End if (bs)

    if (ls)
    {

```



```

// |-----|
// | 2 |.4.....|
// | |.....|
// |-----|
// | 3
// |
// |
// |
// |
// |-----|
//
// IntersectBox4 (blockminxi, blockminyi, blockmaxxi, minyi-1);
// Rect 1
// IntersectBox4 (blockminxi, minyi, minxi-1, maxyi);
// Rect 2
// IntersectBox4 (blockminxi, maxyi+1, blockmaxxi,
blockmaxyi); // Rect 3
    } // End if (rs) tall
    else
    {
        // Wide
        //
        // |-----|
        // | 1 | 2 |
        // | | |
        // | |---|
        // | |.3.|
        // | |...|
        // | |---|
        // | | 4 |
        // |-----|
        //
        IntersectBox4 (blockminxi, blockminyi, minxi-1,
blockmaxyi); // Rect 1
        IntersectBox4 (minxi, blockminyi, blockmaxxi, minyi-1);
// Rect 2
        IntersectBox4 (minxi, maxyi+1, blockmaxxi,
blockmaxyi); // Rect 4
    } // End if (rs) wide
    } // End if (rs)

    break;

case 4:
    // We will make 5 new ones
    //
    if (!flipping || ItsTall(box))
    {
        // Tall
        //
        // |-----|
        // | 1
        // |-----|
        // | 2 |.3.....| 4
        // | |.....|
        // |-----|
        // | 5
        // |

```

```

// |
// |
// -----
//
IntersectBox4 (blockminxi, blockminyi, blockmaxxi, minyi-1);
// Rect 1
IntersectBox4 (blockminxi, minyi, minxi-1, maxyi);
// Rect 2
IntersectBox4 (maxxi+1, minyi, blockmaxxi, maxyi);
// Rect 4
IntersectBox4 (blockminxi, maxyi+1, blockmaxxi,
blockmaxyi); // Rect 5
}
else
// if ((blockmaxyi - blockminyi) <= (blockmaxxi - blockminxi))
{
// Wide
// -----
// | 1 | 2 | 5 |
// | | ----- |
// | | .3..... |
// | | ..... |
// | | ----- |
// | | 4 |
// | |
// -----
//
IntersectBox4 (blockminxi, blockminyi, minxi-1,
blockmaxyi); // Rect 1
IntersectBox4 (minxi, blockminyi, maxxi, minyi-1);
// Rect 2
IntersectBox4 (minxi, maxyi+1, maxxi,
blockmaxyi); // Rect 4
IntersectBox4 (maxxi+1, blockminyi, blockmaxxi,
blockmaxyi); // Rect 5
}
break;

} // End switch (countincluded)
}

```

```

void ViewCookie::IntersectBox4 (int minx, int miny, int maxx, int maxy)
{
    uRect box;
    box.tl.x = minx;
    box.tl.y = miny;
    box.br.x = maxx;
    box.br.y = maxy;

    IntersectBox1 (&box);
}

```

```

// Cooked
// GenCookie.h
void ViewCookie::IntersectBox1(uRect *box)

```

004210"92252950

```

{
    int block;

    if (box->br.x > finalx)
        box->br.x = finalx; // Clip to image

    if (box->br.y > finaly)
        box->br.y = finaly; // Clip to image

    block = FindBlockIntersectingBlock (box);

    if (block < 0) // No intersecting block found.
    {
        AddRectangle (box->tl.x, box->tl.y, box->br.x, box->br.y, 2); // Not
yet ready.
        return;
    }
    else
        FactorBlock (box);
}

```

```

// Block
static void BlockLegend (int xval, int yval, int id, unsigned char *sid)
{
    sprintf ((char *)sid, "%d", id);

    ::TextFont (geneva);
    ::TextSize (10);
    ::MoveTo (xval+8, yval+12);
    ::DrawString (c2pstr((char *)sid));
}

```

```

// Block
void ViewCookie::RenderRect (uRect *r)
// Box around the whole frame, drawn on the first-last row-col of the frame
{
    unsigned char sid[4];
    Rect blonk;

    ////////////////////////////////////////
    //
    // First Block out entire underlying area
    //
    if (!r->status)
        ::RGBForeColor (&white);
    else
        if (r->status == 1)
            ::RGBForeColor (&gray);
        else
            ::RGBForeColor (&dark);

    blonk.left    = r->tl.x;
    blonk.right   = r->br.x+1;
    blonk.top     = r->tl.y;

```

004220"9225960

[illegible]

```
BlockLegend (r->tl.x, r->tl.y, r->id, sid);
}
```

```
// Block
/* Proc */ void ViewCookie::DrawAllBoxes ()
{
    ::PenNormal(); // MQ
    ::GetPort (&secondPort); // MQ

    // First blonk out underlying area.
    ::PenMode (patCopy); // MQ
    ::TextMode (srcOr); // MQ

    //////////////////////////////////////
    //
    // First Block out entire underlying area
    //
    ::RGBForeColor (&white);
    PaintRect (&blonkall);
    everdrawn = 0; // Reset for rubberbanding

    ::TextMode (srcOr); // MQ
    ::RGBForeColor (&black);

    RenderRect (&image);

    //////////////////////////////////////
    //
    // Now do individual rectangles
    //
    int i; // Block
    unsigned char finished = 0;
    i = first;

    if (first < 0)
        return;

    // 0-terminated linked list.
    while (!finished)
    {
        RenderRect (&rlist[i]);
    }
}
```

```

        i = rlist[i].next;
        if (i < 0)
            finished = 1;
    }
}

/* Proc */ void ViewCookie::DrawSelf()
{
    DrawAllBoxes ();
}

// Basic Mouse IO. 1 of 5.
// Cooked.
/* Proc */ void ViewCookie::ClickSelf (const SMouseDownEvent &inMouseDown)
{
    unsigned short Sep = 0; // Default; may be overridden

    ToggleDoDraw ();
}

// Basic Mouse IO. 2 of 5.
// Raw
/* Proc */ void ViewCookie::EventMouseUp (const EventRecord &inMacEvent)
{
    uRect box;

    box.tl.x = xp;
    box.tl.y = yp;
    box.br.x = xp + panw - 1;
    box.br.y = yp + panh - 1;
    box.id = 0;
    box.status = 0;
    box.next = 0; // We don't care; it's a singleton.

    ToggleDoDraw();
    IntersectBox1(&box);
    DrawAllBoxes();
    ReceiveOrderedBoxes();
}

// GenTonal.h
/* Proc */ void ViewCookie::DrawDotBox ()
{
    if (!secondPort)
        return;

    everdrawn = 1;

    ::SetPort (secondPort);    // MQ Test this 980112

    ::PenNormal(); // MQ
    ::RGBForeColor (&black);

```

0042/0" 93252960

```

Point varpt;

::PenMode (patXor);      // MQ
::PenPat (&qd.gray);

::MoveTo (view.tl.x, view.tl.y);
::LineTo (view.br.x, view.tl.y);
::LineTo (view.br.x, view.br.y);
::LineTo (view.tl.x, view.br.y);
::LineTo (view.tl.x, view.tl.y);

::PenPat (&qd.black);
}

/* Proc */ void ViewCookie::HandleDotBox ()
// Cooked
{
    if (everdrawn)
        DrawDotBox (); // First to erase. Always draws at oldxp, oldyp
    oldxp = xp;
    oldyp = yp;
    view.tl.x = oldxp;
    view.tl.y = oldyp;
    view.br.x = oldxp+panw-1;
    view.br.y = oldyp+panh-1;
    DrawDotBox (); // Then to draw
}

// Basic Mouse IO. 4 of 5.
// Cooked
/* Proc */ void ViewCookie::AdjustCursorSelf (Point inPortPt, const EventRecord
&inMacEvent)
{
    xp = inPortPt.h;
    yp = inPortPt.v;

    if (DoDrawBlock)
        HandleDotBox();
}

// Basic Mouse IO. 5 of 5.
// Cooked
/* Proc */ void ViewCookie::SpendTime(const EventRecord& /*inMacEvent*/)
{
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/
//
// Start class WinCookie
//
/* Proc */ WinCookie::WinCookie (MessageT ident,

```

004220"9225250

```

Str255 caption,
LWindow *Caller, // WARNING! Bug of
10/27/97. This would be the more general // LCommander *, but
if so then the procedure screws up the // address.
LCommander *inSuperCommander,
GeneralData *gen,
Boolean ComingFromListener)
: LDialogBox
(WIND_Block,
// (WIND_pickerS,
// (WIND_Tonal,
// windAttr_Modal +
windAttr_Regular +
windAttr_Enabled +
windAttr_Targetable +
windAttr_EraseOnUpdate,
inSuperCommander
),
but_cancel__ (0),
but_accept__ (0),
but_revert__ (0)

{
Int32 groupleft;
int initval;
double initdbl;
int i;

capp__ = inSuperCommander;
gen__ = gen;

for (i = 0; i <= caption[0]; i++)
    mcaption[i] = caption[i]; // mcaption = caption.

SPaneInfo tpi;
SViewInfo tvi;
tvi.imageSize.width = wwidth;
tvi.imageSize.height = 800; // 460; // 330
tvi.scrollPos.h = 0;
tvi.scrollPos.v = 0;
tvi.scrollUnit.h = 1;
tvi.scrollUnit.v = 1;
tvi.reconcileOverhang = true;
tvi.imageSize.width = wwidth;

tpi.visible = true;
tpi.enabled = true;
tpi.bindings.left = false;
tpi.bindings.right = false;
tpi.bindings.top = false;
tpi.bindings.bottom = false;
tpi.superView = this;
tpi.userCon = (Int32)capp__; // pass along ptr to App
tpi.left = tpi.top = 0;
tpi.width = wwidth;
tpi.height = 800; // 460; // 330

```



```
// =====
// File: GenCookie.h
// Copyright, 1999, 2000, Xippix, Inc.
//
// =====
```

```
#pragma once
```

```
#include <LDialogBox.h>
#include <LPane.h>
#include <LStdControl.h>
#include <LGroupBox.h>
```

```
#include "AGAPP.h"
```

```
#include "NumScrollBox.h"
#include "DblScrollBox.h"
class GeneralData; // #include "GeneralData.h"
```

```
class WinCookie;
```

////////////////////////////////////

11

// Contents

11

```
// 1. class ViewCookie : public LView, public LBroadcaster, public LPeriodical
```

```
// 2. class WinCookie : public LDialogBox, public LBroadcaster
```

```
class ViewCookie : public LView, public LBroadcaster, public LPeriodical
```

{

public:

ViewCookie

```
(const SPaneInfo &inPaneInfo,  
const SViewInfo &inViewInfo,  
unsigned short yoffset,  
WinCookie *dlgBox,  
LCommander *inSuperCommander,  
GeneralData *gen);
```

```
// ~ViewCookie();
```

```
typedef struct
```

1

```
int x,y;
```

```
    } uPoint;
```

```
typedef struct
```

1

```
int status, id, next;
```

```
uPoint tl, br;
```

```
    } uRect;
```

```
void RenderRect (uRect *thisrect);
```



```

    unsigned char bitmap[96][12];
};

```

```

class WinCookie : public LDialogBox, public LBroadcaster
{
public:
    WinCookie (MessageT ident,
               Str255 caption,
               LWindow *Caller, // WARNING! Bug of 10/27/97. This would be the
more general // LCommander *, but if so then the procedure
screws up the // address when called.
               LCommander *inSuperCommander,
               GeneralData *gen,
               Boolean ComingFromListener);

    ~WinCookie();

    void ClearPorts();
    void ListenToMessage (MessageT inMessage,void *ioParam);

    LCommander *capp__;
    GeneralData *gen__;

    ViewCookie *secondview__;

    LStdCheckBox *cb_brightness__;
    LStdCheckBox *cb_contrast__;

    LStdButton *but_cancel__;
    LStdButton *but_accept__;
    LStdButton *but_revert__;

    Boolean mComingFromListener;

    Str255 mcaption;
};

```

00420"9232350